

به نام خدا



داده کاوی و یادگیری ماشین

تمرین دوم

استاد درس :

جناب آقای دکتر زارع

دستیاران استاد :

سرکار خانم حسنی

آقای داوردوسن

آقای اسماعیلی

حسین رفیعی زاده

830400027

(1.1

```
df=pd.read_csv("Auto.csv")
x=df.iloc[:,1:8]
x[ "bias"]=1
```

فایل CSV را از ورودی میخوانیم و ستون های یک تا هشت آن را با استفاده از تابع iloc میخوانیم و یک ستون دیگر به نام بایاس به آن اضافه میکنیم.

	cylinders	displacement	horsepower	...	year	origin	bias
0	8	307.0	130	...	70	1	1
1	8	350.0	165	...	70	1	1
2	8	318.0	150	...	70	1	1
3	8	304.0	150	...	70	1	1
4	8	302.0	140	...	70	1	1
..
392	4	140.0	86	...	82	1	1
393	4	97.0	52	...	82	2	1
394	4	135.0	84	...	82	1	1
395	4	120.0	79	...	82	1	1
396	4	119.0	82	...	82	1	1

Print(x)

```
x=np.array(x)
scaler=MinMaxScaler()
scaler.fit(x)
x=scaler.transform(x)
```

داده ها را با استفاده از کتابخانه `sklearn` نرمال سازی میکنیم. (طبق گفته خانم حسنی برای نرمال سازی داده ها و `split` کردن داده ها مشکلی نیست از کتابخانه `sklearn` کنیم)

```
In [113]: runfile('C:/Users/Hossein/q1_gra.py', wdir='C:/Users/Hossein')
[[1.        0.61757106 0.45652174 ... 0.        0.        0.      ]
 [1.        0.72868217 0.64673913 ... 0.        0.        0.      ]
 [1.        0.64599483 0.56521739 ... 0.        0.        0.      ]
 ...
 [0.2       0.17312661 0.20652174 ... 1.        0.        0.      ]
 [0.2       0.13436693 0.17934783 ... 1.        0.        0.      ]
 [0.2       0.13178295 0.19565217 ... 1.        0.        0.      ]]
```

داده ها در بازه 0 تا 1 قرار گرفتند.

```
y=np.array(df[['mpg']])
scaler=MinMaxScaler()
scaler.fit(y)
y=scaler.transform(y)
```

Mpg را میخواهیم predict کنیم پس ستون mpg را در متغیر y میریزیم تا بعدا به راحتی بتوانیم به آن دسترسی داشته باشیم، سپس داده هایش را نرمال سازی میکنیم.

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
lx_train=len(x_train)
```

داده ها رو split کردیم به 80% داده های آموزشی و 20% داده های تست

```

def sgd(data,params,epochs,batchsize) :

    for i in range(epochs):
        for batch in iterate_minibatches(x_train, y_train, batchsize):
            x_batch, y_batch = batch
            lx_batch=len(x_batch)
            Thetas=np.zeros(8)

            for j in range(lx_batch):
                for k in range(8):
                    Thetas[k]+= (1/lx_batch+1)*((x_batch[j]*params).sum()- y_batch[j])*x_batch[j][k]
                    alpha=1/(1+Thetas[k])
                    params=params - alpha*Thetas
                    print(cost(data,params))
    return(params)

```

تابع stochastic gradient descent که دارای چهار ورودی می باشد که اولین ورودی آن داده های آموزشی می باشند و ورودی های بعدی شامل آرایه پارامتر ها (یا همان تتا ها) و تعداد دور ها و batch size می باشد.

در گرادیان کاهشی ما باید هر بار باید پارامتر ها را آپدیت کنیم تا زمانی که که مقدار تابع هزینه ما به حداقل برسد و در گرادیان کاهشی ما روی همه ی داده ها عمل train را انجام میدهیم ولی در گرادیان کاهشی تصادفی هر بار بر روی زیر مجموعه ای از داده ها انجام میشود و سپس پارامتر ها آپدیت می گردند. در اینجا هم به همین صورت می باشد داده آموزشی همراه با batchsize به تابع iterate_minibatches پاس داده شده اند تا زیر مجموعه از داده ها را به ما بدهند. سپس با تغییر دادن مقادیر پارامتر ها سعی داریم تا مقدار تابع هزینه را کم کنیم. بعد از اینکه وزن ها را پیدا کردیم از طریق الفا که همان نرخ یادگیری می باشد مکان مان را آپدیت میکنیم که تا به جایی که حداقل هزینه را دارد برسیم. در ضمن بروز رسانی مقدار تتا ها نباید به صورت ترتیبی قرار گیرد باید به صورت همزمان باشد

```

#runing Gradiant descent
params=np.zeros(8)
epochs=100
batchsize=32
params=sgd(x_train,params,epochs,batchsize)
print(params)

```

مقادیر را شامل تعداد دور ها و batchsize و داده آموزشی را به تابع SGD پاس دادیم.

```

def iterate_minibatches(inputs, targets, batchsize, shuffle=False):
    assert inputs.shape[0] == targets.shape[0]
    if shuffle:
        indices = np.arange(inputs.shape[0])
        np.random.shuffle(indices)
    for start_idx in range(0, inputs.shape[0] - batchsize + 1, batchsize):
        if shuffle:
            excerpt = indices[start_idx:start_idx + batchsize]
        else:
            excerpt = slice(start_idx, start_idx + batchsize)
        yield inputs[excerpt], targets[excerpt]

#cost function
def cost(data,params):
    total_cost=0
    for i in range(lx_train):
        total_cost+=(1/lx_train) * ((data[i]*params).sum()-y_train[i])**2
    return total_cost

```

تابع هزینه که مجموع مربعات خطای محاسبه میکند و تابع `iterate_minibatches` که دسته های کوچک و تصادفی از نمونه ها را پیدا میکند.

```

[0.01865024]
[0.01863048]
[0.01861084]
[0.01859131]
[0.01857191]
[0.01855263]
[0.01853347]
[0.01851443]
[0.01849551]
[0.01847671]
[0.01845803]
[0.01843947]
[0.01842104]
[0.01840272]
[0.01838452]
[0.01836644]
[0.01834848]
[0.01833063]
[0.01831291]
[0.0182953]
[0.01827781]
[0.01826043]
[0.01824317]
[0.01822603]
[0.018209]
[0.01819209]
[0.01817529]
[0.0181586]
[0.01814202]
[0.01812556]
[0.01810921]
[0.01809297]
[-0.01711915 -0.12584614  0.27201622 -0.09889516  0.24955411  0.34043012
 0.24778472  0.          ]

```

همانطور که مشاهده میشود مقدار تابع هزینه دارد کم می شود این کار را تکرار میکنیم تا وقتی که همگرا شوند یعنی اینکه به نقطه ای برسیم که مقدار قبل و جدید تنا با هم مساوی شوند و زمانی مساوی میشوند که گرادیان برابر صفر شود و مشتق زمانی صفر هست که نقاط در \max یا \min هستند ولی در اینجا تعداد تکرارها طبق صورت سوال برابر 100 میباشد. در اینجا مقدار نرخ یادگیری (α) برابر 0.999 می باشد که به نسبت زیاد چون اگر مقدار α زیاد باشد قدم های خیلی بزرگ بر میداریم و ممکن است از روی نقطه بهینه پرش کنیم و اگر مقدار α خیلی کوچک باشد گرادیان کاهشی به کندی همگرا خواهد شد و زمان زیادی میبرد.

```

def predict(x_test):
    Y = (1+(params[0] * x_test[0]) + (params[1] * x_test[1]) + (params[2] * x_test[2]) + (params[3] * x_test[3]) +
        (params[4] * x_test[4]) + (params[5] * x_test[5]) + (params[6] * x_test[6]) + (params[7] * x_test[7]))

    return Y

```

مقادیر تتا ها را در داده تست ورودی ضرب میکنیم تا مقدار y یا همان mpg را به ما بدهد.

$$Y = 1 + \theta_1 x + \theta_2 x + \theta_3 x + \theta_4 x + \theta_5 x + \theta_6 x + \theta_7 x$$

```

y_pred=predict(x_test[0])
mse = np.mean((y_test[0] - y_pred)**2)
print(mse)

```

حالا نوبت به محاسبه MSE Error میرسد X_test[0] را به ورودیتابع predict داده ایم تا مقدار y آن کند سپس از طریق y پیش بینی شده و y واقعی طبق فرمول MSE Error را محاسبه میکنیم.

0.847980060601714

MSE Error for linear

(1.2

```
def predict(x_test, params):
    y=1
    for i in range (len(x_test)-1):
        y += params[i]*(x_test[i] **2)
    return y
```

طبق صورت سوال گفته شده است که چند جمله ای باید درجه 2 باشد برای پیاده سازی رگرسیون چند جمله ای با درجه 2 کافی هست عناصر هر سطر که برابر با مقدار ویژگی هاست به توان دو برسانیم سپس در مقدار پارامترها ضرب کنیم.

```
y_pred=predict(x_test[0] , params)
mse = np.mean((y_test[0] - y_pred)**2)
print(mse)
```

حالا MSE را محاسبه میکنیم.

```
In [137]: runfile('C:/Users/Hossein/q1 gra.py', wdir='C:/Users/Hossein')
0.6283286666303962
```

MSE Error for nonlinear

همانطور که مشاهده می شود مقدار MSE برای غیر خطی کمتر شد که نشان میدهد که اگر تابع `predict` چند جمله ای از درجه 2 باشد مقداری نزدیک تر به مقادیر واقعی حدس میزنند و دقیق تر از خطی می باشد.

(1.3

$$\theta = \theta - \eta \operatorname{diag}(G)^{-\frac{1}{2}} \circ g$$

$$\theta_j = \theta_j - \frac{\alpha}{\sqrt{G_{j,j}}} g_j.$$

الگوریتم AdaGrad طبق فرمول های بالا عمل میکند.

ϵ = Fudge_Factor

η = stepsize

```
def adagrad(data,params,epochs,batchsize,fudge_factor = 0.0001,stepsize = 0.001) :  
    for i in range(epochs):  
        for batch in iterate_minibatches(x_train, y_train, batchsize):  
            x_batch, y_batch = batch  
            lx_batch=len(x_batch)  
            slopes=np.zeros(8)  
  
            for j in range(lx_batch):  
                for k in range(8):  
                    slopes[k]+=((x_batch[j]*params).sum()- y_batch[j])*x_batch[j][k]  
  
            params=params - stepsize * math.sqrt(abs(slopes[k]))*slopes  
            print(cost(data,params))  
    return(params)
```

تغیراتی در تابع داده شد طبق فرمول AdaGrad

```

[0.02121066]
[0.0208198]
[0.02047474]
[0.02016898]
[0.01989706]
[0.01965431]
[0.01943681]
[0.01924119]
[0.01906461]
[0.01890465]
[0.01875923]
[0.01862657]
[0.01850514]
[0.01839365]
[0.01829095]
[0.01819607]
[0.01810816]
[0.01802648]
[0.0179504]
[0.01787934]
[0.01781281]
[0.01775039]
[0.01769169]
[0.01763637]
[0.01758412]
[0.0175347]
[0.01748785]
[0.01744336]
[0.01740105]
[0.01736074]
[0.01732228]
[-0.00984447 -0.04594362  0.05839615 -0.05256609  0.36422025  0.37300894
 0.17570939  0.          ]
0.7926236283200406

```

مقادیر وزن ها وتابع هزینه

```

y_pred=predict(x_test[0])
mse = np.mean((y_test[0] - y_pred)**2)
print(mse)

```

محاسبه مقدار MSE در الگوریتم AdaGrad

```
In [165]: runfile('C:/Users/Hossein/untitled4.py', wdir='C:/Users/Hossein')
0.8140510173520332
```

MSE Error for Linear

```
In [232]: runfile('C:/Users/Hossein/Adagrad.py', wdir='C:/Users/Hossein')
0.484055629821792
```

MSE Error for polynomial

(1.4

```
def Momentum(data,params,epochs,batchsize,l_rate , alpha) :
    for i in range(epochs):
        for batch in iterate_minibatches(x_train, y_train, batchsize):
            x_batch, y_batch = batch
            lx_batch=len(x_batch)
            Thetas=np.zeros(8)
            v=0.0
            for j in range(lx_batch):
                for k in range(7):
                    params[k]+= (1/lx_batch)*((x_batch[j]*params).sum()- y_batch[j])*x_batch[j][k]

                    beta=random.uniform(0,1)
                    v=beta*v
                    params=params - alpha*(beta*v+l_rate*params[k])
                    print(cost(data,params))
    return(params)
```

تغیراتی در تابع داده شد طبق فرمول Momentum میدهیم

```
params=np.zeros(8)
epoch=100
batchsize=32
l_rate = 0.01
alpha=0.1
params=Momentum(x_train,params,epoch,batchsize,alpha,l_rate)
print(params)
```

مقدار دهی به متغیر و فراخوانی تابع Momentum

```
[0.15622661]
[0.15580746]
[0.15538921]
[0.15497188]
[0.15455545]
[0.15413994]
[0.15372534]
[0.15331165]
[0.15289887]
[0.15248701]
[0.15207605]
[0.15166601]
[0.15125687]
[0.15084865]
[0.15044134]
[0.15003494]
[0.14962945]
[0.14922487]
[0.1488212]
[0.14841845]
[0.1480166]
[0.14761567]
[0.14721565]
[0.14681654]
[0.14641834]
[0.14602105]
[0.14562467]
[0.1452292]
[0.14483465]
[0.144441]
[0.02462451 0.01560526 0.0167144 0.0193513 0.02655389 0.03544699
 0.02046124 0.          ]
```

مقادیر وزن ها و تابع هزینه

```
In [234]: runfile('C:/Users/Hossein/Momentum.py', wdir='C:/Users/Hossein')
0.31718418460600717
```

MSE Error for polynomial

```
In [235]: runfile('C:/Users/Hossein/Momentum.py', wdir='C:/Users/Hossein')
0.3510893507414336
```

MSE Error for Linear

(2.1

```
df=pd.read_csv("Weekly.csv")
x=df.iloc[0:,1:7]
```

فایل CSV را از ورودی میخوانیم و ستون های یک تا هفت آن را با استفاده از تابع ilock میخوانیم.

```
x=np.array(x)
scaler=MinMaxScaler()
scaler.fit(x)
x=scaler.transform(x)
```

داده را با استفاده از کتابخانه sklearn نرمال سازی میکنیم.

```
y=np.array(df[['Direction']])

for data in range(len(y)):
    if(y[data]=='Up'):
        y[data]=1
    else:
        y[data]=0
```

ستون Direction که میخواهیم آن را از Predict کنیم، مقدار آن را از دیتا فریم میخوانیم و در متغیر Y میگذاریم حالا مقدار آن را به integer تبدیل میکنیم هر جا که Up بود 1 میگذاریم و هر جا که down بود صفر قرار میدهیم.

```

def sgd(data,params,lrate,epochs,batchsize) :

    for i in range(epochs):
        for batch in iterate_minibatches(x_train, y_train, batchsize):
            x_batch, y_batch = batch
            lx_batch=len(x_batch)
            slopes=np.zeros(6)
            beta=random.uniform(0,1)
            v=0.0
            v=beta*v
            alpha=0.1

            for j in range(lx_batch):
                for k in range(6):
                    slopes[k]+= (1/lx_batch)*((x_batch[j]*params).sum()- y_batch[j])*x_batch[j][k]
            params=params -alpha*(beta*v+lrate*params)
            print(cost(data,params))
    return(params)

```

توضیحات مانند سوال اول هست میخواهیم در تابع `sgd` مقدار وزن ها را به دست بیاوریم که در این سوال ما باید شش وزن را به دست بیاوریم تا بتوانیم از طریق آن ها مقدار `y` را `predict` کنیم.

```

params=np.zeros(6)
lrate = 0.01
epochs = 200
batchsize=32
params=sgd(x_train,params,lrate,epochs,batchsize)

```

مقدار دهی اولیه به متغیر ها و فراخوانی تابع `sgd`

$$f(x) = \frac{1}{1 + \exp(-0.5x)} - 0.5 + u = 0$$

تابع Robbins-Monro به این صورت می باشد.

```
def predict(x_test):
    u=random.uniform(0.0, 0.1**2)
    y = 1
    for i in range(len(x_test)):
        y += params[i] * x_test[i]
    return (1.0 / (1.0 + exp(-0.5*y)))-0.5+u
```

داده تست را از ورودی میگیرد و برای u یک عدد Random در نظر میگیرد و سپس از طریق پارامتر ها و تابع predict مقدار خروجی را Robbins-Monro میکند.

```
[0.3914069508343771]
[0.39074145821808]
[0.39007876758357973]
[0.38941886785639274]
[0.38876174800459656]
[0.38810739703866554]
[0.3874558040113138]
[0.3868069580173324]
[0.3861608481934305]
[0.38551746371807605]
[0.3848767938113387]
[0.3842388277347302]
[0.38360355479104685]
[0.3829709643242154]
[0.3823410457191361]
[0.381713788401526]
[0.3810891818377653]
[0.38046721553474017]
[0.3798478790396969]
[0.3792311619400784]
[0.3786170538633797]
[0.378005544476993]
[0.3773966234880549]
[0.37679028064329784]
[0.37618650572889833]
[0.37558528857033]
[0.3749866190322112]
[0.3743904870181577]
[0.37379688247063514]
[0.373205795370812]
[0.3726172157384087]
[0.3720311336315608]
[0.37144753914666107]
[0.3708664224182227]
[0.06099889 0.06869694 0.06404551 0.06828235 0.06454241 0.01874561]
```

مقادیر وزن ها در 200 بار تکرار و مقادیر تابع هزینه که در هر تکرار با تغییر مقادیر تناها تابع هزینه دارد کمتر می شود.

(2.3

```
def Momentum(data,params,epochs,batchsize,l_rate , alpha) :  
    for i in range(epochs):  
        for batch in iterate_minibatches(x_train, y_train, batchsize):  
            x_batch, y_batch = batch  
            lx_batch=len(x_batch)  
            Thetas=np.zeros(6)  
            v=0.0  
            for j in range(lx_batch):  
                for k in range(6):  
                    params[k]+=(1/lx_batch)*((x_batch[j]*params).sum()- y_batch[j])*x_batch[j][k]  
  
                    beta=random.uniform(0,1)  
                    v=beta*v  
                    params=params - alpha*(beta*v+l_rate*params[k])  
                    print(cost(data,params))  
    return(params)
```

Momentum تابع

```
[2.1670258820997524e+107]  
[8.424451617555625e+108]  
[3.2750594094320376e+110]  
[1.2732002772687885e+112]  
[4.9496474517952136e+113]  
[1.9242070815140766e+115]  
[7.480478011026833e+116]  
[2.908083636685579e+118]  
[1.1305361001652825e+120]  
[4.395031345225092e+121]  
[1.7085965253729592e+123]  
[6.6422781937339e+124]  
[2.5822281005354468e+126]  
[1.0038576778499245e+128]  
[3.9025608820897056e+129]  
[1.5171454853078935e+131]  
[5.898000039291156e+132]  
[2.292885211098844e+134]  
[8.913737803073396e+135]  
[3.465272541221621e+137]  
[1.34714684729591414e+139]  
[5.237119471493198e+140]  
[2.0359636664885022e+142]  
[7.91493887780155e+143]  
[3.0769830754093313e+145]  
[1.196196836454312e+147]  
[4.650291654116286e+148]  
[1.8078306018968913e+150]  
[7.028057008557584e+151]  
[2.7322020804221574e+153]  
[1.0621610210579732e+155]  
[4.129218855146317e+156]  
[1.6052602209703225e+158]  
[-3.99257034e+78 -4.20156736e+78 -4.04067155e+78 -4.17880034e+78  
-4.15201734e+78 -1.02235810e+78]
```

وزن ها و مقادیر تابع هزینه